Hoisting Nested Functions

Developer: Haroon Khaleeq <haroon.khaleeq@stud.tu-darmstadt.de>
Pranay Sarkar <pranay.sarkar@stud.tu-darmstadt.de>

Supervisor: Marina Billes <marina.billes@crisp-da.de>
Software Lab -SOLA
FB 20 Informatik

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Program Testing & Analysis Project work
Winter 2016-17
Fachbereich Informatik (FB 20)

Contents

1 Abstract

JavaScript is widely used for writing client-side web applications and is getting increasingly popular for writing mobile applications. JavaScript is over fifteen years old; nevertheless, the language is still misunderstood by what is perhaps the majority of developers and designers using the language. One of the most powerful, yet misunderstood, aspects of JavaScript are functions. The ability of defining a function inside another function and passing function as an argument of another function provides more flexibility to the programmer to build higher order function which intern supports higher abstraction. But misuse of this prower can lead to bad coding standard. While terribly vital to JavaScript, their misuse can introduce inefficiency and hinder an application's performance. The Google's V8 JavaScript engine nested version (functions inside other functions) is 42% slower than the equivalent version without this nesting. Thus we present results of a dynamic analysis aiming to detect these functions which can be hoisted and defined outside the inner function it is declared. This helps reducing unnecessary stack space. Unlike C, C++, and Java, there are not that many tools available for analysis and testing of JavaScript applications. In this report, we are going to present a dynamic analysis for identifying these nested function (hoistable functions) using simple yet powerful framework, called Jalangi. Other than reporting the hoistable function out analysis also results the total number functions that were executed, not executed, not hoistable and outer functions. Further the analysis is evaluated by running on well known libraries of node.js namely *underscore*, *lodash*, *q* . We believe that all no false positives were reported; hence the analysis is quite efficient and effective.

2 Introduction

Today's JavaScript engines are light-years ahead of the engines of ten years ago, but they do not optimize everything. What they don't optimize is left to developers. This is pushing JavaScript developers to ensure their code is efficient and has a good performance. The problem with nested functions is one characteristic of JavaScript that hinders performance: the nested function is repeatedly created due to repeated calls to the outer function. The misuse of functions can introduce inefficiency and hinder an application's performance.

The ability to define a nested function addresses new programming construct which provides more flexibility and power to programmer. but if misused the advantage it provides can turn to a disadvantage. nested function helps programmer to use in-line function call and to define singleton function which has their own advantages. It also provides ability to create higher order functions which can be passed as arguments and returned as value. This new programming construct is useful when we need more abstraction and reuse in lower level. Using function expressions to define a function inside another function gives the ability to define a function dynamically and determine which function to use programmatically. The ability to access a variable defined in the same scope an inner function is defined promotes the programming construct known as clause. Clauses also have an advantage in web based programming. But it depends on the situation which programmer has to decide whether to use nested functions. It is a performance overkill both in terms of performance and memory consumption if the nested function are defined in a loop or defined inside a function which is called several times.

3 Hoisting in JavaScript

It should be mentioned that JavaScript have function-level scope instead of block-level scope like other languages. That brings 'Hoisting' into the scope of discussion.

3.1 What is Hoisting

In JavaScript a name enters in scope in four different ways:

- **Language - defined:** All scopes are, by default, given the names *this* and *arguments*
- **Formal parameters:** Functions that have named formal parameters, scoped to the body of the function
- **Function decelerations:** Anything in the form of *foo() {}*
- **Variable declarations:** These take the form *var foo;*

Function and variable decelerations are always moved invisibly to the top of their containing scope by the JavaScript interpreter. This is called Hoisted property of JavaScript. For example, this piece of code:

```
function foo() {
  bar();
  var x = 50;
}
```

is interpreted like this:

```
function foo() {
 var x;
 bar();
 x = 50;
}
```

3.2 Nested functions in JavaScript

Functions definitions inside other functions are allowed in JavaScript. These kind of functions are called nested functions. Nested functions are created every time outer function is called. For example:

```
var x = 23;
function f(a){
 function g(step){
  return x + step;
 }
 g(a);
}
f(2);
```

Here the function *g(step)* does not depend on the surrounding function *f(a)*

3.3 Hoisted functions in JavaScript

The given nested function in the last example can be converted into Hoisted function in the following way:

```
var x = 23;
function g(step){
 return x + step;
}
function f(a){
 g(a);
}
f(2);
```

3.4 Advantage of Hoisted functions over Nested functions

It is observed that the nested version of the function in the example is **42%** slower than Google's V8 JavaScript engine than the equivalent hoisted version where $g()$ is defined outside of $f()$. It should be mentioned that function's name doesn't get hoisted if it is part of a function expression.

4 Jalangi

The implementation of the analysis for detecting hoistable functions in JavaScript programs is based on Jalangi2. Jalangi is a proven dynamic analysis framework for JavaScript. Jalangi generates an instrumented intermediate code from JavaScript which it used for its analyses. It assigns a unique id sid to each JavaScript loaded in runtime. Each sid is mapped to an object called $iids$. $iids$ is an array which represents each instruction in the code using a iid which stands for instruction id. The iid is a unique value which is assigned to each callback function inserted by Jalangi. $iids$ also maps each iid to an array containing $[beginLineNumber, beginColumnNumber, endLineNumber, endColumnNumber]$. The mapping from $iids$ to arrays is only available if the code is instrumented with the $--inlineIID$ option.

Jalangi has several types of callback functions as mentioned above. Each instruction in a program can be mapped to a callback. While executing the target program using Jalangi the analysis executes the callbacks for each of the instructions. To test a particular behavior of the program the callbacks can be used in the analysis and can be implemented as needed. Jalangi analyses can be executed online using browser or using command line or stored in some log file from command prompt or some runtime script. The provided analysis runs in a command line where the written analysis program and the program on which the analysis is to be performed has to be specified with few other optional parameters to the Jalangi2 executor program.

Now we will discuss in brief the callbacks which are used in the analysis file. Each function with other parameters provides a $-lstinline-iid-$ parameter which can be used to uniquely distinguish the instruction for which the callback is called. The analysis logic uses the following callbacks:

4.1 invokeFunPre(iid, f, base, args, isConstructor, isMethod, functionIid, functionSid)

$invokeFunPre()$ call back in called before a function or method or constructor is invoked. Parameter f provides the function object whose body is going to be invoked. The *name* attribute of the function object can be used to retrieve the function name. This feature is used in the analysis file which will be discussed later. Using the iid parameter the line number of the source code where the function is declared can be obtained.

4.2 invokeFun(iid, f, base, args, result, isConstructor, isMethod, functionIid, functionSid)

Similar to $invokeFunPre()$, this callback is executed after the function or method or constructor invocation.

4.3 declare(iid, name, val, isArgument, argumentIndex, isCatchParam)

This callback is called for every local variable declared in the scope, for every formal parameter, for every function defined using a function statement, for arguments variables, and for the

formal parameter passed in a catch statement. *name* provides the name of the variable declared and val provides the initial value of the variable at declaration time. $isArgument$ parameter is $true$ if the variable is a arguments or a formal parameter. $isCatchParam$ parameter is $true$ if the variable is a parameter of a catch statement.

4.4 getField(iid, base, offset, val, isComputed, isOpAssign, isMethodCall)

This callback is called after property of an object is accessed in the function scope. $offset$ provides the name of property. $isComputed$ property is $true$ if the property is accessed using squire brackets. For example, $isComputed$ is $true$ is the operation is $a[b]$ but is false is the operations is $a.b$.

4.5 putField(iid, base, offset, val, isComputed, isOpAssign)

$putField()$ callback is called after a property of an object is written. Just like $getField()$, it also have $offset$ which provides the name of property, and $isComputed$ does the same thing.

4.6 read(iid, name, val, isGlobal, isScriptLocal)

Read call back is called after a variable is read. *name* parameter provides the name of the variable being read. val parameter provides the value of the variable. $isGlobal$ is $true$ if the variable is not declared using var keyword in the current script. $isScriptLocal$ parameter becomes $true$ if the variable is declared in the global scope using var.

4.7 read(iid, name, val, isGlobal, isScriptLocal)

Write callback is called before a variable is written. the meaning of the parameters are same as read callback.

5 Approach to the Goals

5.1 Goals

Main goals of the project are:

1. Dynamic analysis of the JavaScript file done using Jalangi framework.
2. At first the dynamic analysis should fins all nested functions.
3. Then developing a dynamic analysis to detect all functions that can be hoisted or are already hoisted.

5.2 Approach

We have designed our dynamic analysis where we are using $ProgramStack$ for storing all information fetched using Jalangi callbacks. $FunctionList$ contains all the functions found during the dynamic test. $FunctionsIDs$ array consist of all function ID fetched during the analysis. The dynamic analysis the following things:

5.2.1 Step 1: Identifying the nested functions

Jalangi analysis identifies which functions are nested functions. The following Jalangi hooks are used for that purpose: *invokeFunPre()*, *invokeFun()*, *declare()*. This following function *get_nested_functions()* is used to fetch a list of all nested functions inside a function.

5.2.2 Step 2: Identifying variable's usage scope

As per the definition of Hoisted functions, the function can not use any variable declared in the parent scope of that function. We approach the problem in this following way:

1. Identify all variables that is being written to, in the parent scope and make a list of them.
2. Identify all variables that is being read in the child or nested function.
3. If there is any such variable which is written to in the parent scope and is read from the nested child function, then the child function is not a hoisted function or it can not be hoisted. That function is excluded from the final list of hoisted functions.
4. The same steps are followed for the all properties of the all used objects.

These following Jalangi callbacks are used for this purpose: *getField()*, *putField()*, *read()*, *write()*.

5.2.3 Step 3: Identifying functions with same name

Jalangi analysis tries to identify if there is already a function having same name as nested function in any scope, starting from the global scope. This function *get_nf_not_globally_declared()* returns a list of all nested functions which does not exist globally with the same function name (i.e. no duplicate function names).

5.2.4 Special Step: Anonymous Function Handling

It is observed that while handling Anonymous functions, $invokeFunPre()$ does not return any $f.name$. This normally creates problem in identifying function name from the used $programstack$. It is also observed that the variables to which anonymous functions are assigned, those corresponding names can be used as the anonymous function name in the $programstack$. So the analysis takes the last accessed variable into consideration before the anonymous function invocation and stores the name as a function name in the program stack. When the end of anonymous function block is encountered, analysis does the same by putting the variable name to mark the ending block of the anonymous function in program stack. The function $check_for_anonymous_functions()$ takes care of all these mentioned steps.

6 Corner Cases

During the dynamic analysis of hoisted functions with Jalangi we happen to come across some cases which can not be detected properly using Jalangi 2 framework. Some of the corner cases are described here.

6.1 Anonymous Functions

..

6.2 Recursive Functions

..

6.3 Property of object

If there are different properties of different object, Jalangi 2 can not properly detect the base object of a particular property. For example, in a program if we have

```
var b.a = 23;
var c.a = 44;
```

Differentiating between those 2 a's is not possible with Jalangi predefined callback functions.

7 Conclusions

We have developed a dynamic analysis which was tested on three node.js libraries. The hoistable functions detected in all cases where checked manually in the source code and no false positives were found. The analysis also provides additional information on the functions which can not be hoisted or not hoistable.